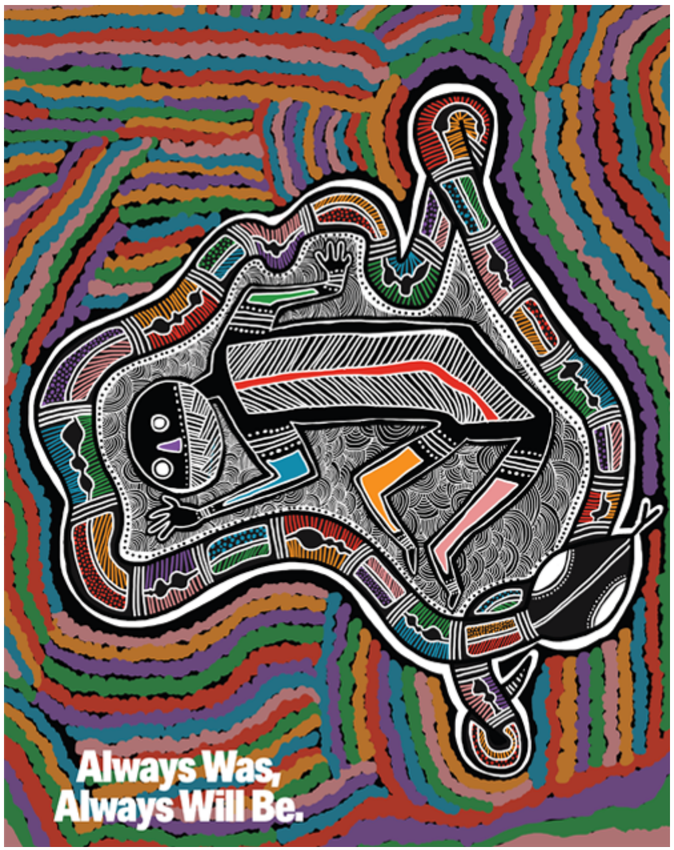
MONASH University

ECE3073

Real Time Systems
Utilisation Statistics
and Why do we need them?

Acknowledgement of Country



We wish to acknowledge the people of the Kulin Nations, on whose land Monash University operates.

We pay our respects to their Elders, past and present.

WARNING

MONASH University

Starting Multitasking

```
void main (void)
{
INT8U err;
OSInit();

/* ---------- Application Initialization ---------- */
OSMutexCreate(9, &err);
OSTaskCreate(TaskPrio10, (void *)0, &TaskPrio10Stk[999], 10);
OSTaskCreate(TaskPrio15, (void *)0, &TaskPrio15Stk[999], 15);
 OSTaskCreate(TaskPrio20, (void *)0, &TaskPrio20Stk[999], 20);
 /* ---------- Application Initialization ---------- */

OSStart();
}
```

Starting Multitasking(2)

```
void  main(void)
        {
        OsInit()
         OSTaskCreateExt(TaskStart,  …
         OSStart();
        }
```

A related context for systems.

There are three working scenarios to consider:

1) Need to know
2) Want to know
3) Access to knowledge

Utilisation

- The study of, and application, of scheduling algorithms requires knowledge of how much work the CPU does.
- The measure is “utilisation”….what percentage of the CPU power is used.
- How do we know the total utilisation and the utilisation of individual tasks?

Utilisation

- We can calculate it from the code and simulation.
- We can measure it directly.
- We can obtain it indirectly during execution by collecting statistical information about how much free time the CPU has available.

Calculating Utilisation

- Run your task codes through a simulator and from the total execution cycles you have a measure of each task utilisation.
- Summing this for all tasks and knowing system clock speed you can obtain the total utilisation and, therefore, the CPU load.

Measuring Utilisation

- With our tasks running on a target processor we can measure the execution time of a task (just as in the lab tests)
- For periodic tasks we know how often they need to run so the task utilisation is simply (Execution time)/Period
- For aperiodic tasks we need to know the worst case (highest) execution rate for the task

Statistical Evaluation

- How do we know how much spare time the CPU has?
- Collect data on how much time it is not doing anything useful………the idle time
- The problem…..processor clock speed and timers compared with “real time”.
- How do we determine this in a processor independent fashion?

Timer Interrupt

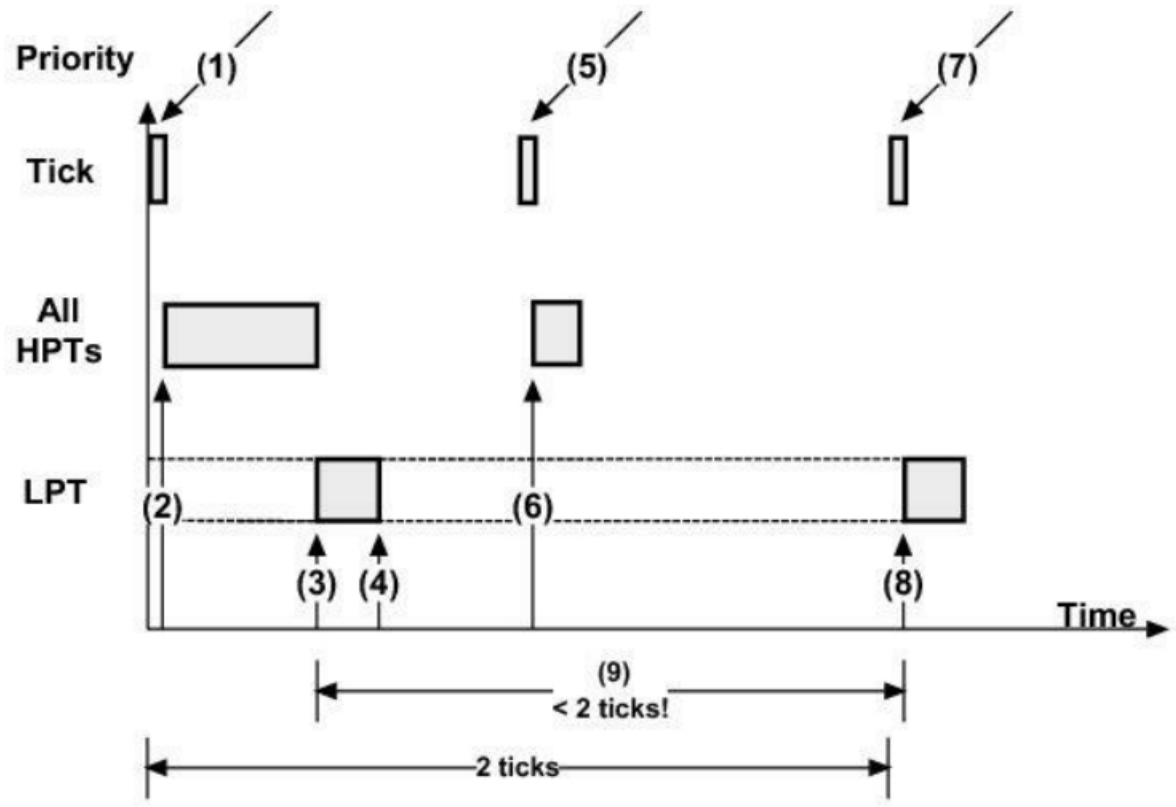
- **Why does an RTOS need a timer interrupt?**
  - Sleeping: zzzzzz….
  - Waking: !!!!!!!!!!
  - Timeouts: whooops, give up
  - Keeping track of time: tick tock tick tock ….

Using the timer interrupt

- **OSTimeDlyHMSM(h,m,s,ms)**
  - Delay task for specified time
- **OSTimeDly(n)**
  - Delay task for specified number of ticks (timer interrupts)

uC OSTimeDly(n)

- **Actual time delay may vary:**



MONASH University

Collecting Statistics

- In an RTOS designed to be portable across many processors this is not an easy job.
- The designer of uCOSII (Jean Labrosse) solved it in a rather elegant way but to use it he needed a very particular approach.
- He should be proud of it and the next slides go through the process.
- To use it you just need to ensure you use the required approach.

uC OS- II Idle Task

- The **Idle task runs when no other task is ready**
- The **Idle task is the lowest priority task.**
- **Allows CPU usage to be measured using the counter OSIdleCtr:**

```
void OS_TaskIdle( void *pdata){
    for (;;){
    OS_ENTER_CRITICAL();
            OSIdleCtr++;
    OS_EXIT_CRITICAL();
    OSTaskIdleHook();
    }
}
```

MONASH University

Statistics Task

- **The task OS_TaskStat() runs every second**
- **Calculates integer % CPU usage stored in OSCPUUsage**
- **OSCPUUsage is a global variable, so accessible from any task.**

Statistics Task (cont'd)

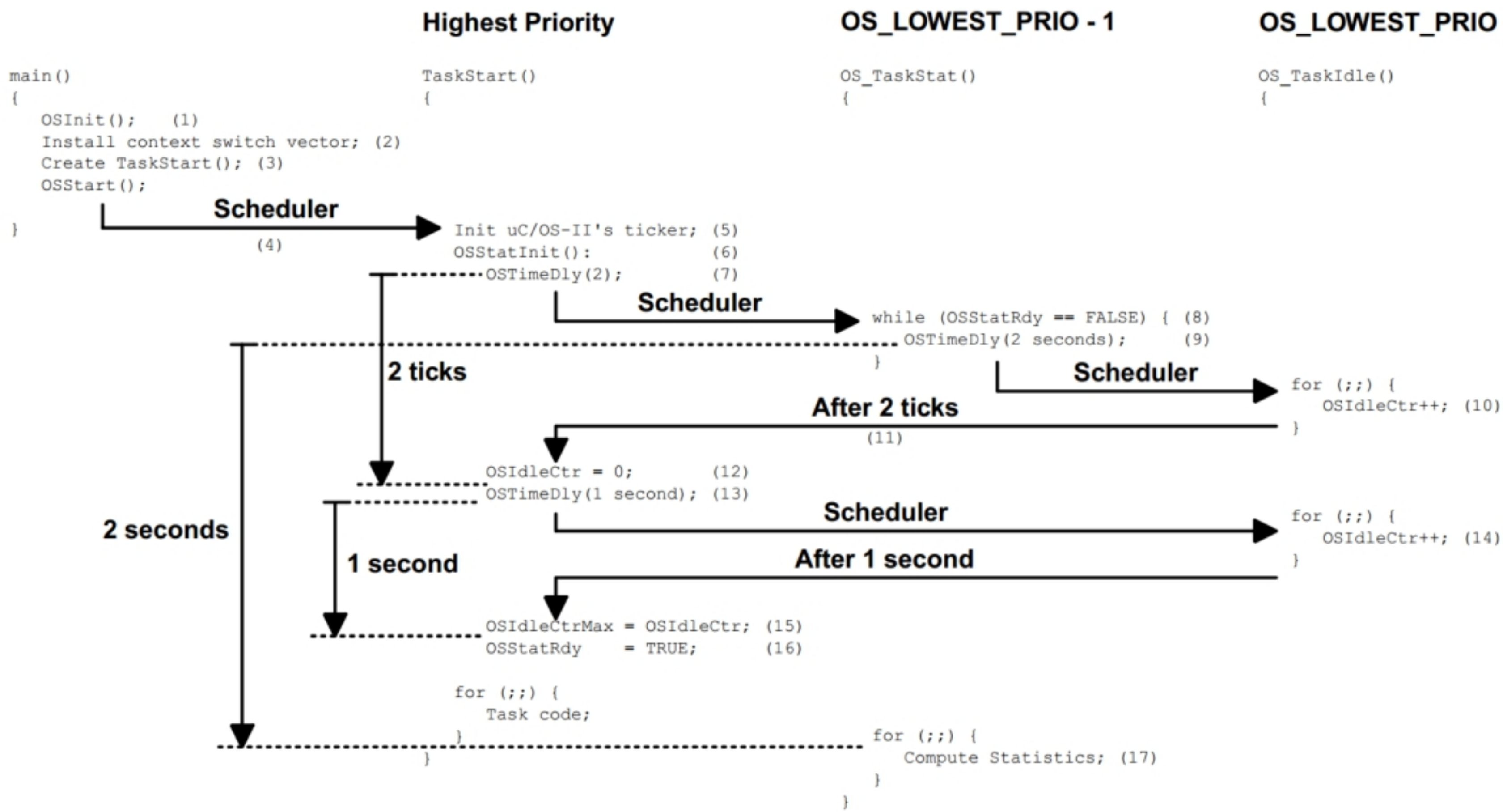
- **To use OSCPUUsage, it must be initialised:**
  - Call OSStatInit() from ***first and only*** task running

```
void main(void){
   // initialisation code here ..
   OSTaskCreateExt(TaskStart, …
   OSStart();
   }
void TaskStart(void *pdata){
   OSStatInit();
   // create other tasks and implement …
   }
```

STATISTICS TASK INITIALISATION

```
void OSStatInit (void){
    // must run when no other tasks running
    // Finds OSIdleCtr for one second
    OSTimeDly(2);
                // now synchronised to timer interrupt
    OS_ENTER_CRITICAL();
                OSIdleCtr = 0L;
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);
    OS_ENTER_CRITICAL();
                OSIdleCtrMax = OSIdleCtr;
                OSStatRdy = TRUE;
    OS_EXIT_CRITICAL();
} // OSStatInit
```

MONASH University



MONASH University

How it works!!!
Study these two slides carefully

F3.9(1) The first function that you must call in µC/OS-II is `OSInit()`, which initializes µC/OS-II.

F3.9(2) Next, you need to install the interrupt vector that performs context switches. Note that on some processors (specifically the Motorola 68HC11), you do not need to install a vector because the vector is already resident in ROM.

F3.9(3) You must create `TaskStart()` by calling either `OSTaskCreate()` or `OSTaskCreateExt()`.

F3.9(4) After you are ready to multitask, call `OSStart()`, which schedules `TaskStart()` for execution because it has the highest priority.

F3.9(5) `TaskStart()` is responsible for initializing and starting the ticker. You want to initialize the ticker in the first task to execute because you don't want to receive a tick interrupt until you are actually multitasking.

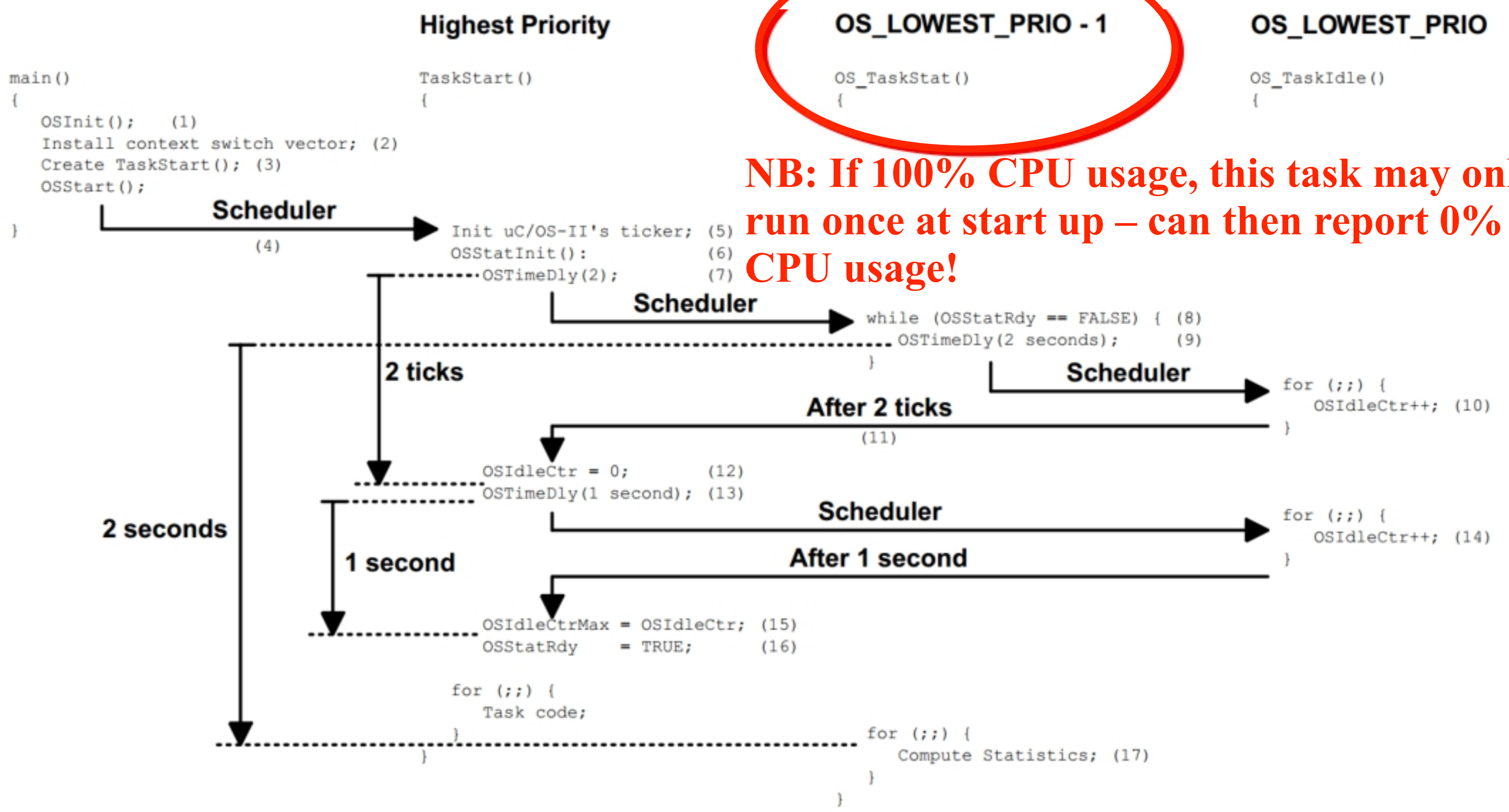
F3.9(6) Next, `TaskStart()` calls `OSStatInit()`. `OSStatInit()` determines how high the idle counter (`OSIdleCtr`) can count if no other task in the application is executing. A Pentium II running at 333MHz increments this counter to a value of about 15,000,000. `OSIdleCtr` is still far from wrapping around the 4,294,967,296 limit of a 32-bit value. At the rate processor speeds are getting, it will not be too long before `OSIdleCtr` overflows. If overflow becomes a problem, you can always introduce some software delays in `OSTaskIdleHook()`. Because `OS_TaskIdle()` really doesn't execute any useful code, it's OK to throw away CPU cycles.

F3.9(7) `OSStatInit()` starts off by calling `OSTimeDly()`, which puts `TaskStart()` to sleep for two ticks. This action is done to synchronize `OSStatInit()` with the ticker. µC/OS-II then picks the next highest priority task that is ready to run, which happens to be `OS_TaskStat()`.

F3.9(8) The code for `OS_TaskStat()` is discussed later, but as a preview, the very first thing `OS_TaskStat()` does is check to see if the flag `OSStatRdy` is set to `FALSE` and then delays for two seconds if it is.

How it works!!!
Study these two slides carefully

F3.9(9)	It so happens that OSStatRdy is initialized to FALSE by OSInit(), so OS_TaskStat() in fact puts itself to sleep for two seconds. This action causes a context switch to the only task that is ready to run, OS_TaskIdle().
F3.9(10)	The CPU stays in OS_TaskIdle() until the two ticks of TaskStart() expire.
F3.9(11)	
F3.9(12)	After two ticks, TaskStart() resumes execution in OSStatInit(), and OSIdleCtr is cleared.
F3.9(13)	Then, OSStatInit() delays itself for one full second. Because no other task is ready to run, OS_TaskIdle() again gets control of the CPU.
F3.9(14)	During that time, OSIdleCtr is continuously incremented.
F3.9(15)	After one second, TaskStart() is resumed, still in OSStatInit(), and the value that OSIdleCtr reached during that one second is saved in OSIdleCtrMax.
F3.9(16)	
F3.9(17)	OSStatInit() sets OSStatRdy to TRUE, which allows OS_TaskStat() to perform a CPU usage computation after its delay of two seconds expires.



NB: If 100% CPU usage, this task may only run once at start up – can then report 0% CPU usage!

Computing CPU Usage

- **When non idle tasks run, *IdleCtr* cannot increment.**
- **(*IdleCtrMax – IdleCtr*) used to calculate *CPUUsage* every second in OS_TaskStat() *:***

$$\mathit{OSCPUUsage\%} = 100 \times \left(1 - \frac{\mathrm{OSIdleCtr}}{\mathrm{OSIdleCtrMax}}\right)$$

$$= 100 - \frac{100 \times \mathrm{OSIdleCtr}}{\mathrm{OSIdleCtrMax}} \quad \longleftarrow \textit{Overflow possible}$$

$$= 100 - \frac{\mathrm{OSIdleCtr}}{\left(\frac{\mathrm{OSIdleCtrMax}}{100}\right)}$$

Integer divides now usable

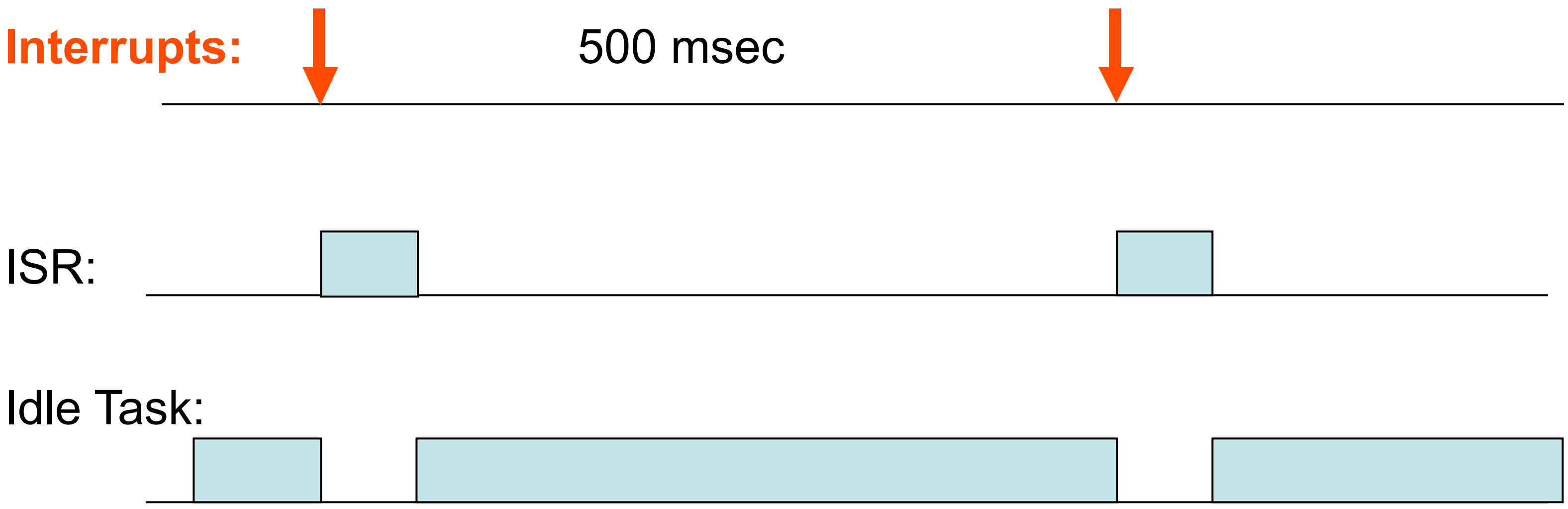
void OS_TaskStat (void *p_arg)

```
void    OS_TaskStat (void *p_arg)
{
  while (OSStatRdy == OS_FALSE) {
        OSTimeDly(2 * OS_TICKS_PER_SEC);
  }
  OSIdleCtrMax /= 100L;
  for (;;) {
     OS_ENTER_CRITICAL();
     OSIdleCtrRun = OSIdleCtr;
      OSIdleCtr       = 0L;
      OS_EXIT_CRITICAL();
      OSCPUUsage    = (INT8U)(100L - OSIdleCtrRun /
OSIdleCtrMax);
     OSTaskStatHook();
     OSTimeDly(OS_TICKS_PER_SEC)
  }
}
```

MONASH University

Measuring Timer Interrupt Overhead

- **OSIdleCtrMax**
  - Represents one second's worth of the idle task running *with interrupts occurring* .
  - Eg with 2 interrupts per second:



Interrupt Overhead (cont'd)

- **Suppose OSIdleCtrMax reaches Idle2 counts in a second for 2 Hz interrupts**
- **Suppose Idle1 counts for 1 Hz interrupts – measured after reconfiguring system.**

MONASH University

Interrupt Overhead (cont'd)

- **Q: What is the %CPU overhead for the 1Hz timer interrupt?**
- **A: Idle1 – Idle2 represents one ISR time in idle count units.**

  => Idle counts in an UNINTERRUPTED second
  = Idle1 + (Idle1- Idle2)

  %CPU time of One ISR per second is:

  (Idle1 – Idle2)/[Idle1 + (Idle1- Idle2)] * 100

- **Can you generalise this approach to 100 Hz interrupts and 1000 Hz interrupts?**